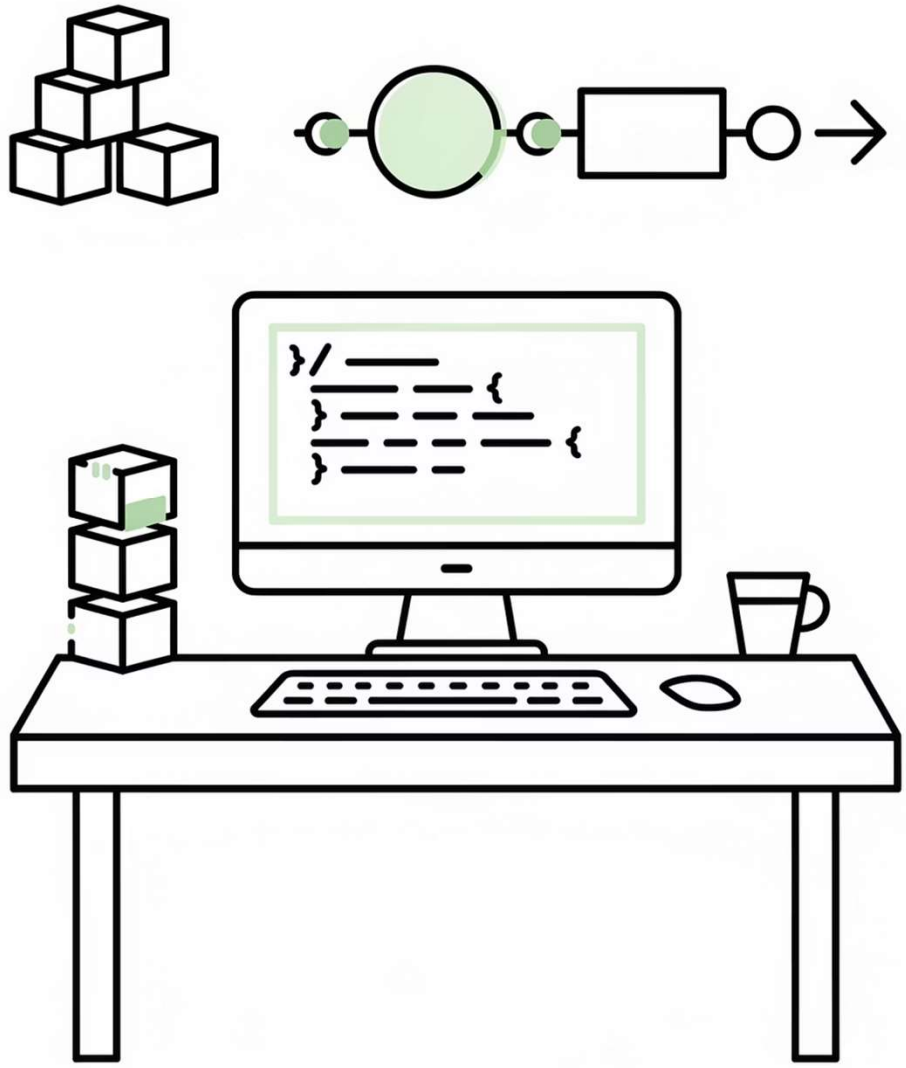
CI/CD with GitHub Actions & Docker

From Code Commit to Container Registry

Agenda Highlights

- CI vs CD
- Docker in pipelines
- GitHub Actions container workflows
- Secure secrets & versioning
- Release strategies



Learning Objectives

By the end of this session, participants will be able to:

- *Differentiate CI vs CD clearly*
- *Build Docker images automatically using GitHub Actions*
- *Push images securely to Docker Hub / ECR / ACR*
- *Use GitHub Secrets safely*
- *Version Docker images using Git tags*
- *Apply basic release strategies*

CI vs CD (Conceptual Clarity)

Continuous Integration (CI)

- Triggered on **code changes**
- Focus: **Build, Test, Validate**
- Fast feedback to developers

Continuous Delivery (CD)

- Artifact is **ready for deployment**
- Deployment may be **manual approval**

Continuous Deployment

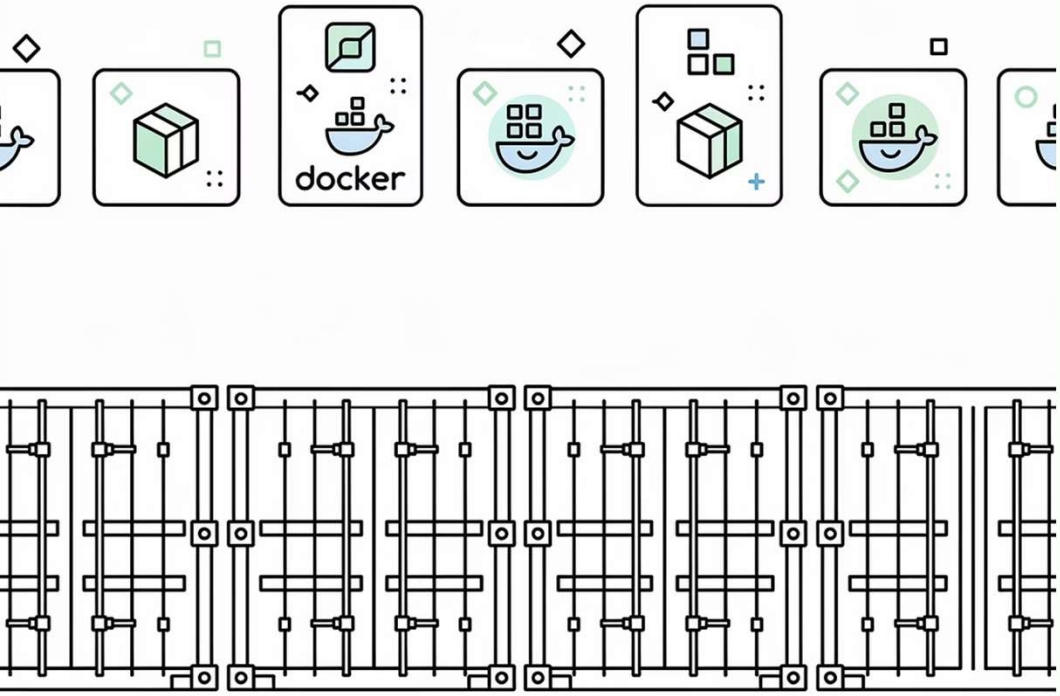
- Fully automated deployment to production

📌 ***Key Trainer Note:***

CI = "Is the code good?" CD = "Is it safely deployable?"

CI vs CD Comparison Table

Aspect	CI	CD
Trigger	Code push / PR	After successful CI
Output	Build artifact	Deployed app
Risk	Low	Medium - High
Automation	Partial	High
Example	Build Docker image	Deploy to Kubernetes



Why Docker in CI/CD Pipelines?

Problems without Docker

- "Works on my machine"
- Environment drift
- Dependency mismatch

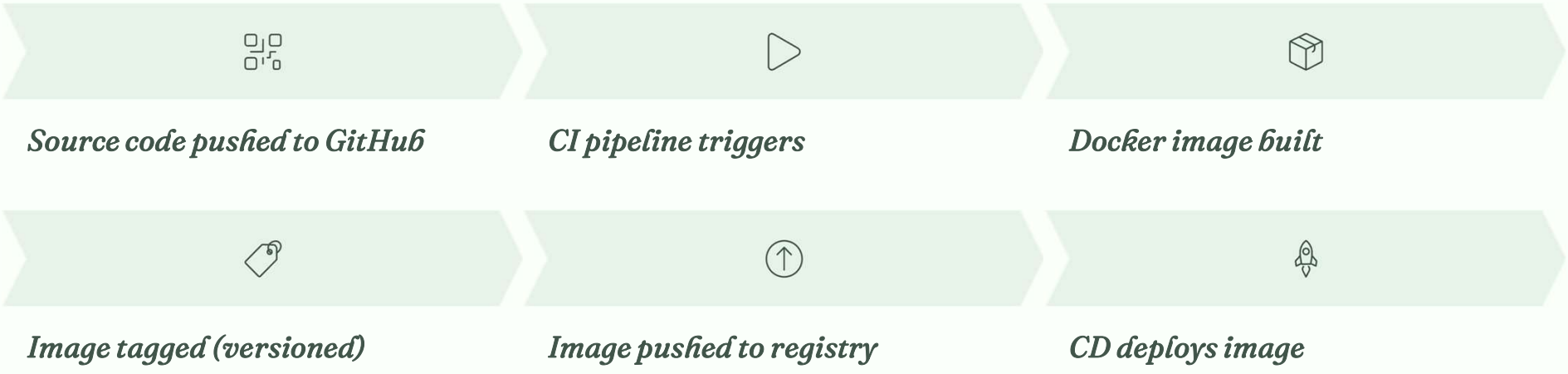
Benefits with Docker

- Immutable artifacts
- Consistent runtime
- Faster deployments
- Easy rollback

📌 Industry Reality:

Modern CI/CD = **Code** → **Container** → **Registry** → **Platform**

Docker Lifecycle in CI/CD



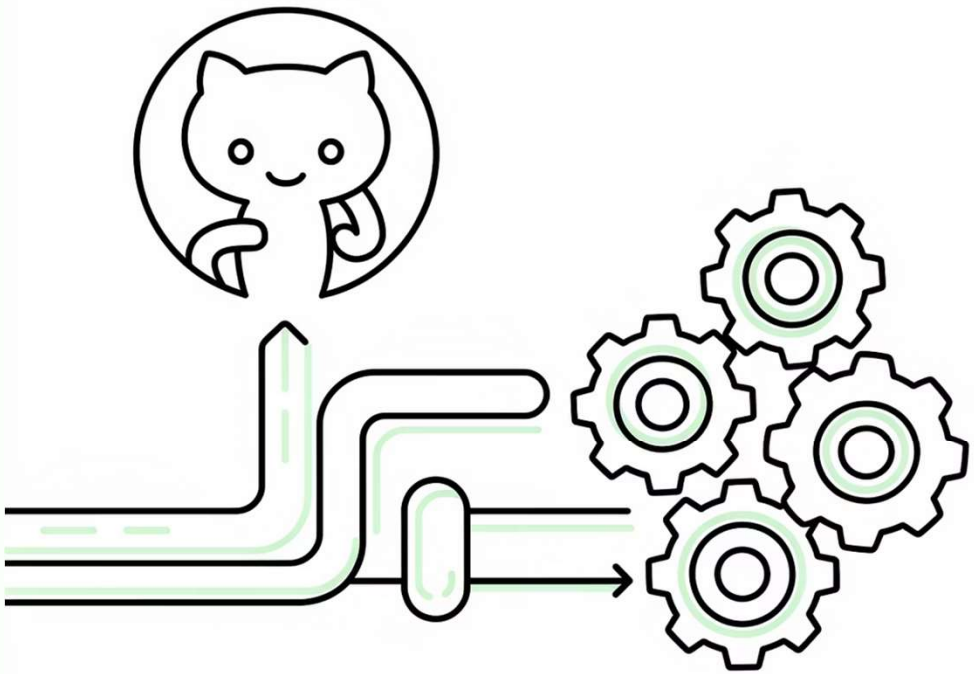
Introduction to GitHub Actions

What is GitHub Actions?

- Native CI/CD service in GitHub
- YAML-based workflows
- Event-driven automation

Why GitHub Actions?

- No external CI tool required
- Tight GitHub integration
- Free minutes for public repos



GitHub Actions Core Components

Workflow

YAML file defining automation

Event

Trigger (push, PR, tag)

Job

Group of steps

Step

Individual action

Runner

VM executing the job

GitHub Actions Workflow Structure

```
name: CI Pipeline
on:
  push:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
```

Explanation

- `on` → trigger
- `jobs` → pipeline stages
- `runs-on` → execution VM
- `steps` → commands/actions

Container Workflow Use Case

Typical Container CI Workflow

01

Checkout code

02

Set up Docker

03

Build Docker image

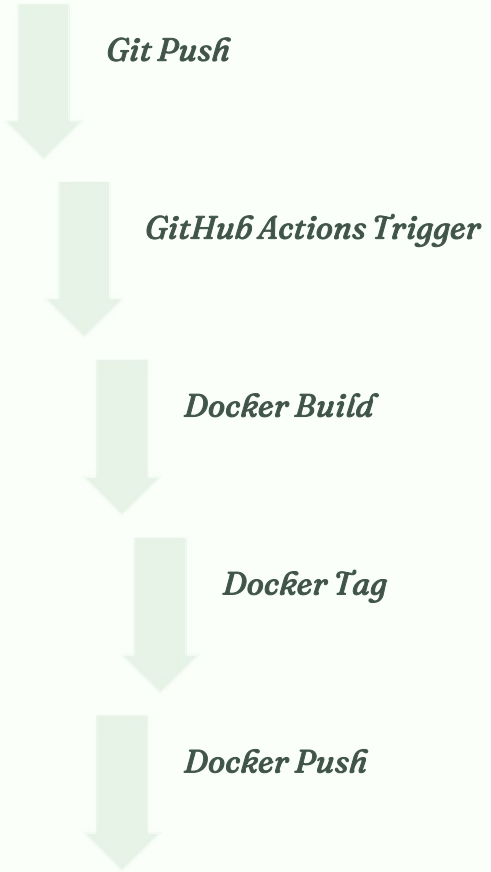
04

Authenticate to registry

05

Push image

Docker Build & Push: High-Level Flow



Sample Project Structure

```
repo/
├── Dockerfile
├── app.py
├── requirements.txt
└── .github/
    └── workflows/
        └── docker-ci.yml
```

📌 ***Trainer Tip:***

Enforce .github/workflows/ convention early.

Dockerfile Example (Code Highlight)

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY app.py .
CMD ["python", "app.py"]
```

Explanation

- Lightweight base image
- Layer caching optimization
- Production-ready structure

GitHub Actions: Docker Build Workflow

```
name: Docker CI
on:
  push:
    branches:
      - main
```

Explanation

- Workflow triggers only on main
- Prevents accidental builds

Checkout & Docker Setup

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout source
        uses: actions/checkout@v4
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3
```

Why Buildx?

- Multi-platform builds
- Faster caching
- Modern Docker standard